LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-682249

Best Practices for Scalable Power Measurement and Control

S. Walker, M. Mcfadden

February 8, 2016

IEEE Workshop HPPAC'16
Chicago, IL, United States
May 23, 2016 through May 27, 2016

Disclaimer

Best Practices for Scalable Power Measurement and Control

Scott Walker, *Student Member, IEEE*, and Marty McFadden
Lawrence Livermore National Lab

***Abstract*—There are thousands of specialized registers on modern processors which provide useful features such as power budgeting, thermal monitoring, and performance counting. These registers fall into two categories, model specific registers (MSRs) and configuration space registers (CSRs). Many of these registers, such as those supported by PAPI, help the high-performance computing (HPC) community analyze their program in order to maximize performance and use resources more efficiently. However, there are many MSRs and CSRs which are not supported by existing performance tools. The MSR kernel module provides access to all MSRs, but requires the user to be root. Users will typically want to access a handful of MSRs sequentially, but we found that the existing MSR module has far too much overhead. Just like MSRs, users need elevated privileges to access CSRs with utilities such as lspci. Simply allowing users to access all of the registers is out of the question because of the security risks involved. Furthermore, using these registers requires detailed knowledge of the architectural changes in addition to the manufacturer's proprietary ways of encoding the data.**

In this paper, we will describe a group of utilities developed at Lawrence Livermore National Laboratory to address these problems. Our Libmsr API solves the usability issues by providing a simplified interface to common tasks. The companion kernel module, MSR-SAFE, allows whitelisting MSRs for userspace access, plus provides an optimized way to access MSRs in batches. Our CSR-SAFE kernel module is a first of its kind utility to whitelist sections of the PCI Express configuration space, where CSRs reside, for userspace access. We demonstrate the abilities Libmsr and MSR-SAFE by using the utilities to set various power limits on the NAS parallel benchmark MG. Experiments analyzing our optimizations show a maximum speedup of 26X for certain MSRs, and about 16X on average for sufficiently large batches versus the stock MSR kernel module. Our utilities currently support most modern Intel processors.

I. Introduction

There are thousands of specialized registers on modern processors which are useful for the high-performance computing (HPC) community. These registers can be used for tasks such as power and performance analysis, which help HPC developers tailor their code to maximize performance and efficiently utilize resources[8][1]. There are two main classifications: Model Specific Registers (MSRs) and Configuration Space Registers (CSRs). Existing utilities such as PAPI provide an interface to the performance counter MSRs [13], however, the majority of MSRs and all CSRs are not supported by existing tools such as PAPI.

Because of security risks and usability issues, having access to all MSRs and CSRs can currently only be accomplished by system administrators with detailed knowledge of Intel processors. Simply changing the permissions on the existing register access utilities is also out of the question, since many MSRs present security risks such as denial of service or privilege escalation. Furthermore, accessing MSRs with the existing module has far too much overhead for many use cases. This is mainly due to the fact that MSRs are accessed one at a time via *pread* and *pwrite* calls. Unlike MSRs, Configuration Space Registers have no general access mechanism. CSRs for specific devices are accessed transparently through device drivers or through utilities such as sysfs and lspci. Another factor limiting the usefulness of MSRs and CSRs is the architectural knowledge required to use them correctly. The directions for using each MSR are embedded in Intel documentation spanning thousands of pages [12][10][9][11]. To make matters worse, MSRs and CSRs can change between architectures, making portability a concern.

In this paper we will provide an overview of a group of utilities developed at Lawrence Livermore National Lab to solve the current problems with accessing MSRs and CSRs. Our kernel modules, MSR-SAFE and CSR-SAFE, provide userspace access to MSRs and CSRs respectively. In addition, we provide a mechanism for administrators to supply a whitelist of registers, thus mitigating the security risks. We also provide an optimized method for accessing a batch of MSRs using an *ioctl* call. Our Libmsr API solves the usability problems by dynamically detecting the architecture and available registers, plus it handles encoding the data properly. Furthermore, common functions such as getting power readings and setting power limits can be done with a few simple Libmsr function calls. Our utilities currently support Intel processors, but we hope to expand the supported hardware in the future.

This paper is organized as follows. Section II will cover how we enable safe userspace access of MSRs and CSRs using a whitelist. Section III will discuss the MSR-SAFE and CSR-SAFE kernel modules and their performance. Section IV will present the various capabilities of the Libmsr API and demonstrate some of the power management functionality. Section V will discuss related work and section VI concludes the paper.

II. Whitelisting Registers

The existing MSR kernel module provides an all or nothing approach to access MSRs. MSR accessibility is severely limited in userspace since many have security risks such as denial of service. For example, there are two registers: APERF and MPERF that are responsible for tasks such as

timekeeping and calculating effective clock frequency. We were able to take down our test system by writing these MSRs at carefully selected times. The Intel documentation states that the processor must receive the *wrmsr* instructions for APERF and MPERF registers sequentially but does not give any indication why. With the existing Intel kernel module this would not even be possible due to the overhead. To provide a way for users to access registers on shared clusters without creating security risks, our kernel modules MSR-SAFE, and CSR-SAFE use a whitelist (see figures 1 and 6). This allows system administrators to select a list of MSRs users are allowed to access at the granularity of individual bits if necessary. The overhead of the whitelisting was measured at around 20-40ns per lookup, which is negligible compared to the latency of a single MSR instruction.

The CSR whitelist is slightly more complex because CSRs are frequently duplicated for different PCIe devices. For example, the Integrated Memory Controller (iMC) performance counters are duplicated in a different 4KB configuration space for each memory channel [10][12]. Therefore, the CSR whitelist provides a method of inheritance so that changes to duplicated whitelist entries must only be made in one place (see figure 2). The whitelisting overhead has slightly more impact on CSRs for small batch sizes, but at a certain batch size a performance limit is hit, around 4 million operations per second (see figure 5). We suspect that this performance bottleneck is at a part of te processor called the UBox, which services all CSR requests [10]. For both whitelists we supply only a writemask. Users are only allowed to read registers that have a writemask, even if it is null.

III. Kernel Modifications

Optimizations– The existing kernel module allows users to read or write an MSR using *pread* or *pwrite* calls. Once this operation is completed, the calling process can continue and access the next MSR. This mechanism works fine when we are, for example, reading a socket-level MSR such as RAPL registers. However, there are many MSRs at the hardware thread level, such as the performance counters. MSR users will often be utilizing multiple MSRs in rapid succession, for example getting the value of a counter off of every logical thread of a node. We tested reading one MSR using the traditional module and found that polling at 250ms used all of the resources of that core. To make matters even worse, we discovered that accessing an MSR on another socket has more than five times the overhead of accessing an MSR on the current hardware thread. Accessing an MSR on the same socket but on a different core has a similar increase in overhead (see figures 3 and 4). We were able to optimize MSR accesses by providing an *ioctl* system call in MSR-SAFE that allows a batch of registers to be specified for increased throughput (see figure 8).

MSR-SAFE Performance– Our optimizations indicated considerable performance increases over the traditional method of accessing MSRs. We tested multiple types of MSR accesses because we noticed high variability depending on the location of the register. We split the types of accesses into four domains listed below from fastest to slowest:

On Thread
Off Thread On Core
Off Thread On Socket
Off Socket

We tested every different batch size in increments of powers of two, up to 256. Overall, our MSR-SAFE *ioctl* operation is significantly faster than the traditional method of accessing MSRs as soon as the batch contains four or more operations. Even with batch sizes of two, our *ioctl* outperforms *pread* and *pwrite* for nearly every situation. Besides the fact that our *ioctl* call was faster overall, there are a couple other trends within our experiments. First, the *pread* and *pwrite* commands have a constant rate of execution. Second, our *ioctl* command has diminishing returns after a certain batch size depending on where the register is located. We suspect that this is because we are hitting a bandwidth limit on the processors. For certain tests such as off thread on socket, we noticed a large variability between MSR operation speeds. To compensate for this, our tests execute a batch one million times. In addition, we executed the program containing the test at least 20 separate times and took the average across all of these parameters. The tests were executed on a 2 socket 18 core Haswell node. Some of the tests were repeated on the Catalyst cluster at Lawrence Livermore National Lab, an Ivy Bridge system. Those results are not shown here because of their similarity to our Haswell test. Furthermore, due to problems with the job scheduler, we were unable to enable hyperthreading and test the “off thread on core" category.

Configuration Space Registers– Using MSR-SAFE as a guideline, a CSR-SAFE kernel module was devised. CSRs are located in PCIe configuration space (see figure 7), and each socket has its own set of CSRs. The Intel documentation lists bus 0 as the location of ‘core’ CSRs and bus 1 as the ‘uncore’ [10]. These are not actual bus numbers, they are indicators of “Intel’s bus 0" and “Intel’s bus 1". When the CSR-SAFE module is loaded, it discovers the actual bus numbers associated with the Intel hardware. Therefore, users need only utilize the bus, device, function, and offset (CSR number) as listed in the Intel Documentation for the whitelist and module calls. There is a 4KB configuration space for each PCIe bus, device, and function. Linux provides a way to access PCIe configuration space for a particular bus, device, and function in kernel space. First a pci_dev struct is populated, then one can use the function pointers contained within that struct to read or write the configuration space of the struct’s bus, device, and function. The pci_dev struct is very large, and dealing with one for every bus, device, and function in our whitelist was not efficient. We opted to memory map the PCIe configuration space for each whitelisted bus, device, and function so CSR-SAFE would have a smaller memory footprint. CSR-SAFE uses an ioremap for every distinct bus, device, and function in the whitelist when the module is loaded. The ioremap call returns an IO pointer which can then be used with the ioread32

```
# MSR          Write Mask            # Comment
0x00000010     0x0000000000000000    # "SMSR_TIME_STAMP_COUNTER"
0x00000017     0x0000000000000000    # "SMSR_PLATFORM_ID"
0x000000C1     0xffffffffffffffff    # "SMSR_PMC0"
0x000000C2     0xffffffffffffffff    # "SMSR_PMC1"
0x000000C3     0xffffffffffffffff    # "SMSR_PMC2"
0x000000C4     0xffffffffffffffff    # "SMSR_PMC3"
0x000000C5     0xffffffffffffffff    # "SMSR_PMC4"
0x000000C6     0xffffffffffffffff    # "SMSR_PMC5"
0x000000C7     0xffffffffffffffff    # "SMSR_PMC6"
0x000000C8     0xffffffffffffffff    # "SMSR_PMC7"
0x000000CE     0x0000000000000000    # "MSR_PLATFORM_INFO"
0x000000E2     0x00000000fe000407    # "MSR_PKG_CST_CONFIG_CONTROL"
0x000000E7     0x0000000000000000    # "SMSR_MPERF"
0x000000E8     0x0000000000000000    # "SMSR_APERF"
0x00000186     0x00000000ffffffff    # "SMSR_PERFEVTSEL0"
0x00000187     0x00000000ffffffff    # "SMSR_PERFEVTSEL1"
0x00000188     0x00000000ffffffff    # "SMSR_PERFEVTSEL2"
0x00000189     0x00000000ffffffff    # "SMSR_PERFEVTSEL3"
0x0000018A     0x00000000ffffffff    # "SMSR_PERFEVTSEL4"
0x0000018B     0x00000000ffffffff    # "SMSR_PERFEVTSEL5"
0x0000018C     0x00000000ffffffff    # "SMSR_PERFEVTSEL6"
0x0000018D     0x00000000ffffffff    # "SMSR_PERFEVTSEL7"
0x00000198     0x0000000000000000    # "SMSR_PERF_STATUS"
0x00000199     0x0000000000000000    # "SMSR_PERF_CTL"
0x0000019A     0x000000000000001f    # "SMSR_CLOCK_MODULATION"
0x0000019B     0x0000000001ffff0f    # "SMSR_THERM_INTERRUPT"
0x0000019C     0x0000000000000aaa    # "SMSR_THERM_STATUS"
0x000001A0     0x0000000000000000    # "SMSR_MISC_ENABLE"
0x000001A2     0x0000000000000000    # "SMSR_TEMPERATURE_TARGET"
0x000001A6     0x0000003fffff8fff    # "SMSR_OFFCORE_RSP_0"
0x000001A7     0x0000003fffff8fff    # "SMSR_OFFCORE_RSP_1"
0x000001AE     0xffffffffffffffff    # "TURBO_RATIO_LIMIT1"
0x000001B0     0x000000000000000f    # "SMSR_ENERGY_PERF_BIAS"
0x000001B1     0x0000000000000555    # "SMSR_PACKAGE_THERM_STATUS"
```

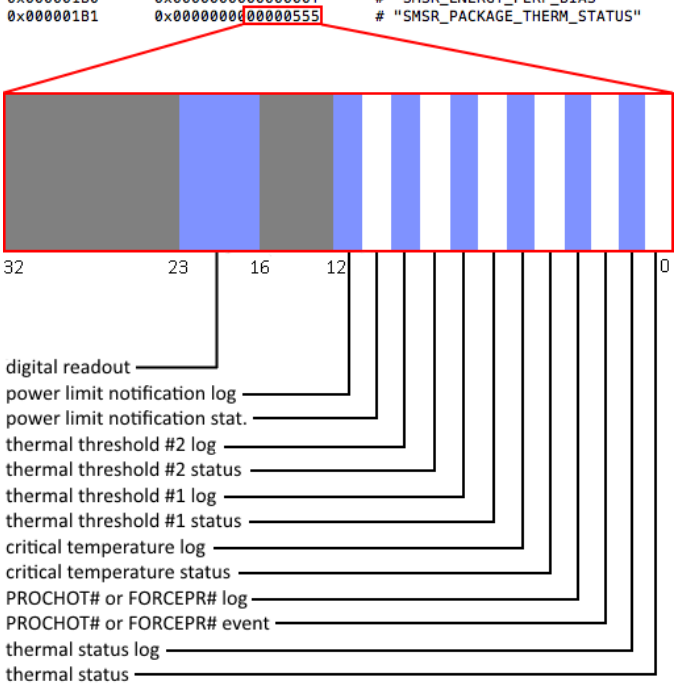
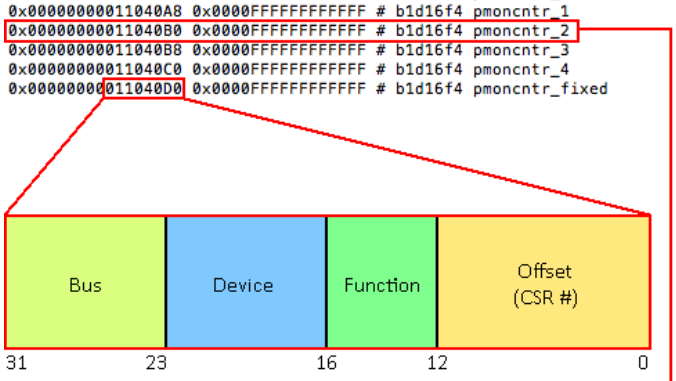


Fig. 1: The MSR-SAFE whitelist format. The gray fields are reserved bits and the blue fields are bits designated as read only by the whitelist. Users may only write bits designated in the writemask (white), but they may read all bits as long as a writemask exists.

and iowrite32 functions. CSR-SAFE keeps track of these IO pointers within the whitelist data structure for fast access. Our initial observations indicated that our memory mapped CSR accesses were faster than using the pci_dev functions. We hope to examine this in more detail in future work. Since there are so many configuration spaces which reuse the CSR numbers as the offset, access with a *pread* or *pwrite* would

```
# CSR              Write Mask         # Comment
# Bus 0 Device 5  Function 0
0x0000000000050000 0x0000000000000000 #b1d5f0 vid
0x0000000000050002 0x0000000000000000 #b1d5f0 did
# Bus 1 Device 16 Function 4
0x0000000001104000 0x0000000000000000 # b1d16f4 vid
0x0000000001104002 0x0000000000000000 # b1d16f4 did
# ...
0x00000000011040A0 0x0000FFFFFFFFFFFF # b1d16f4 pmoncntr_0
0x00000000011040A8 0x0000FFFFFFFFFFFF # b1d16f4 pmoncntr_1
0x00000000011040B0 0x0000FFFFFFFFFFFF # b1d16f4 pmoncntr_2
0x00000000011040B8 0x0000FFFFFFFFFFFF # b1d16f4 pmoncntr_3
0x00000000011040C0 0x0000FFFFFFFFFFFF # b1d16f4 pmoncntr_4
0x00000000011040D0 0x0000FFFFFFFFFFFF # b1d16f4 pmoncntr_fixed
```




```
# ...
# Bus 1 Device 16 Function 5
# Inherits from Bus 1 Device 16 Function 4
0x0110400001105000 0x0000000000000000 # b1d16f4 vid
0x0110400201105002 0x0000000000000000 # b1d16f4 did
# ...
0x011040A0011050A0 0x0000000000000000 # b1d16f5 pmoncntr_0
0x011040A8011050A8 0x0000000000000000 # b1d16f5 pmoncntr_1
0x011040B011050B0 0x0000000000000000 # b1d16f5 pmoncntr_2
```

Fig. 2: The CSR whitelist has additional complexity. The bus, device, function, and offset are encoded into the lower 32 bits of the CSR field. The upper 32 bits encode a previously entered CSR, so that writemasks can be inherited for duplicated CSRs.

not be applicable for a module which accesses multiple PCIe devices. Thus our CSR-SAFE kernel module only makes use of a batch via *ioctl*, where each operation reports the CSR offset, bus, device, function, and socket to the module. The CSRs performance is more consistent than MSRs once we reach the bandwidth limit of CSR accesses, which appears to be about 4 million per second. We attribute this limit to the UBox hardware, which services all CSR accesses for Intel devices [9]. Accessing a single CSR is far faster than accessing a single MSR using the traditional *pread* or *pwrite* mechanism, but the CSR batch hits a performance limit far faster than the MSR batches (see figure 5). Currently, providing raw IO to the whole configuration space is not supported, mainly because of the security risks.

IV. Userspace

The Batch Interface– Libmsr provides a group of functions for easily setting up a batch of MSR operations. There are three stages to set up a batch: user allocation, Libmsr allocation, and batch loading. In the user allocation stage an array of pointers is allocated. To prevent data copying, Libmsr keeps a single copy of the batch MSR data hidden from the user, which can only be accessed through indirection. In the Libmsr

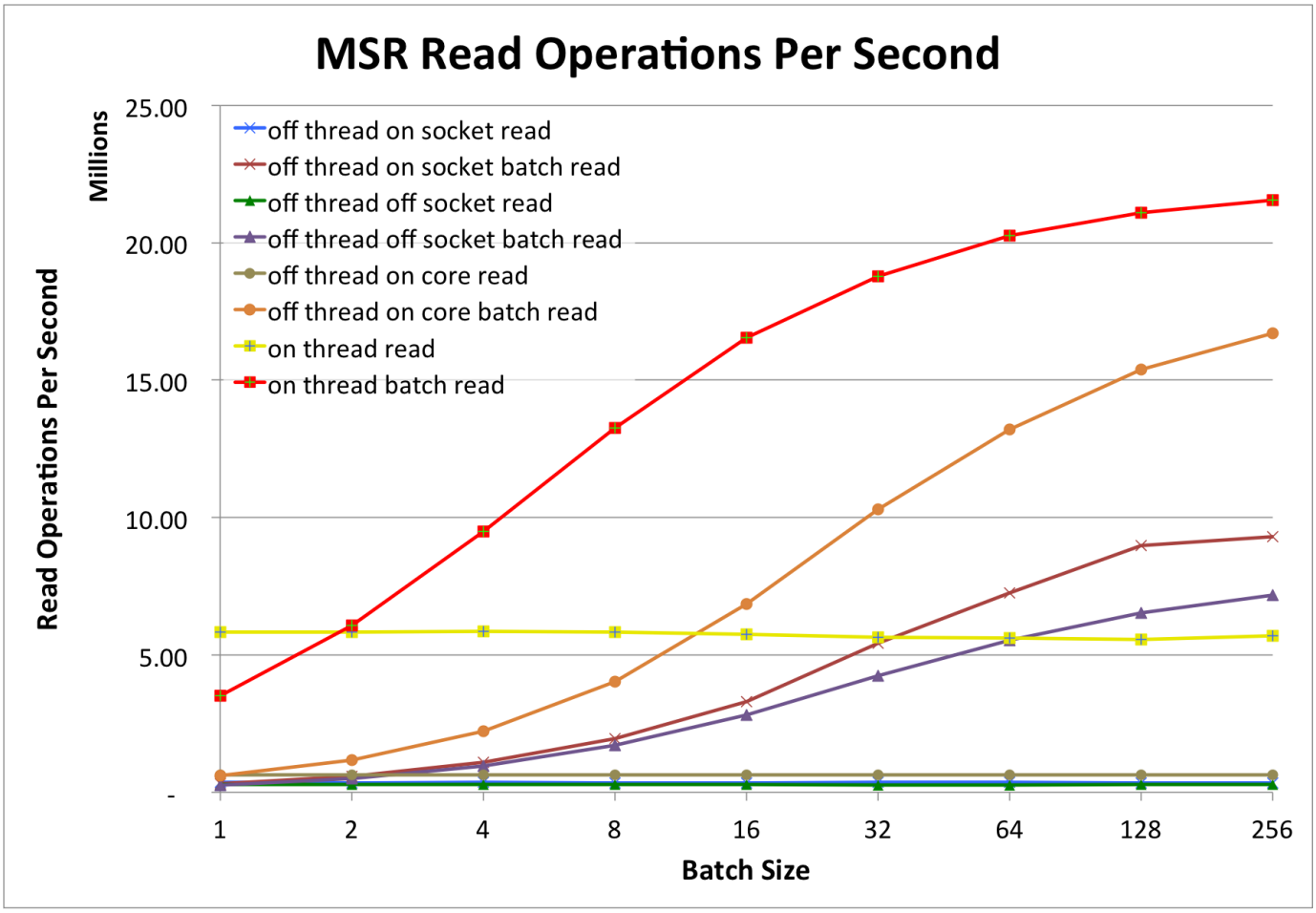


Fig. 3: Measurement of MSR reads per second comparing *pread* to the batch interface *ioctl* on a 2 socket 18 core Haswell node

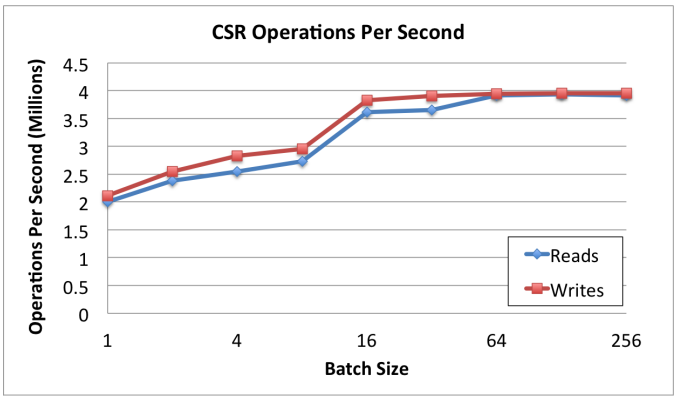


Fig. 7: Measurement of the CSR-SAFE operation speed using *ioctl* on a 2 socket 12 core Ivy Bridge node

allocation stage, the user instructs Libmsr to allocate space for the actual MSR data, passing in parameters indicating the batch they wish to use and the size of the batch. Finally, the batch loading stage tells Libmsr which MSR(s) the user will be accessing with that batch, and links the userspace pointers to the Libmsr data array. After this setup is complete, the user can obtain the data with simple read and write functions.

For CSRs a separate batch interface is provided. Because of the additional complexity of CSRs, a convenient load function is not provided, each operation must be specified. The two allocation stages are the same, but the load stage requires additional information to correctly gather the data. In addition to the CSR number or “offset", users must provide the bus, device, and function number of the register.

RAPL– For tasks such as reading all energy counters or setting limits for a socket, we provide a RAPL interface in Libmsr[11]. By using a struct representing the RAPL power limit, viewing or setting a limit is as simple as a single function call per socket. Similarly, obtaining energy readings only requires calling a function before and after the interval being measured. Libmsr will detect a single energy counter wraparound, so the user need only ensure their interval is not excessively large. Unfortunately we currently do not provide a universal solution for this as this varies greatly between processor models. We may investigate a solution to this in

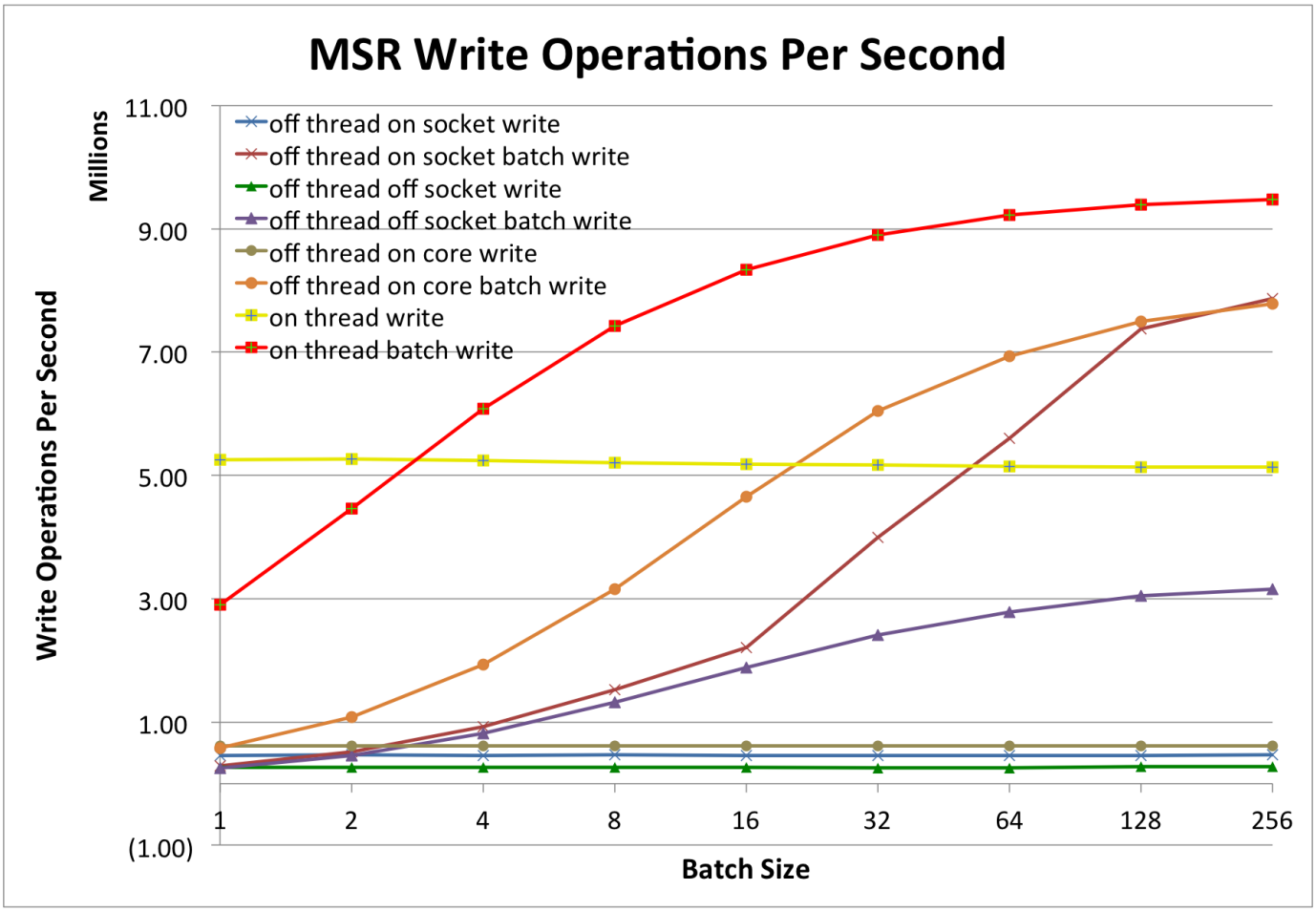


Fig. 4: Measurement of MSR writes per second comparing *pwrite* to the batch interface *ioctl* on a 2 socket 18 core Haswell node

future work. The RAPL interface also contains functions to print or dump the data it has collected, or access the raw data for more advanced purposes. To test Libmsr's implementation of RAPL, we performed an experiment where we set various power bounds with RAPL and recorded subsequent behavior of the NAS parallel benchmark MG (see figure 9). This test was performed on 52 nodes of the Catalyst cluster at Lawrence Livermore National Lab. We observed the execution time vary only slightly with a CPU power cap, which is consistent with the work of Hackenberg et al. [7]. The execution time varied greatly with a memory power cap, as one would expect for a memory bound application like MG [4].

Counters– Libmsr has simplified functions for accessing the many hardware counters on supported architectures. Some of these hardware counters are "fixed" meaning they always count the same event. Other counters require the user to specify what they want it to count. Currently for the non-fixed counters, the user must specify the events to be counted, as specified in the Intel documentation. Unfortunately this highlights a limitation of the library: there are so many counters and events that including them all would be a huge undertaking [11][9][12][10]. Future work may include adding frequently used events.

Clocks– This section of Libmsr contains simple access functions for hardware clocks such as the Timestamp Counter. Because the Timestamp Counter increments at a fixed rate for every processor, it can be used for high resolution timekeeping tasks. The APERF and MPERF clock counters can be used to derive the performance of the processor, where the formula is PercentInUse = PercentNotIdle * (APERF/MPERF) [11]. We suggest that these MSRs be whitelisted as read-only because we were able to create denial of service attacks by writing to these MSRs frequently with random values.

Turbo– The Libmsr Turbo functions can be used to turn Intel Turbo Boost on or off. Our experiments and observations have not indicated a significant difference in power when turbo is on or off. This is consistent with the observations of Hackenberg et al. [7].

Misc– The Misc section of Libmsr provides functions for reading and modifying the MISC_ENABLE register. This register provides a quick way to toggle features such as Fast String, Turbo, and Precise Event Based Sampling (PEBS).

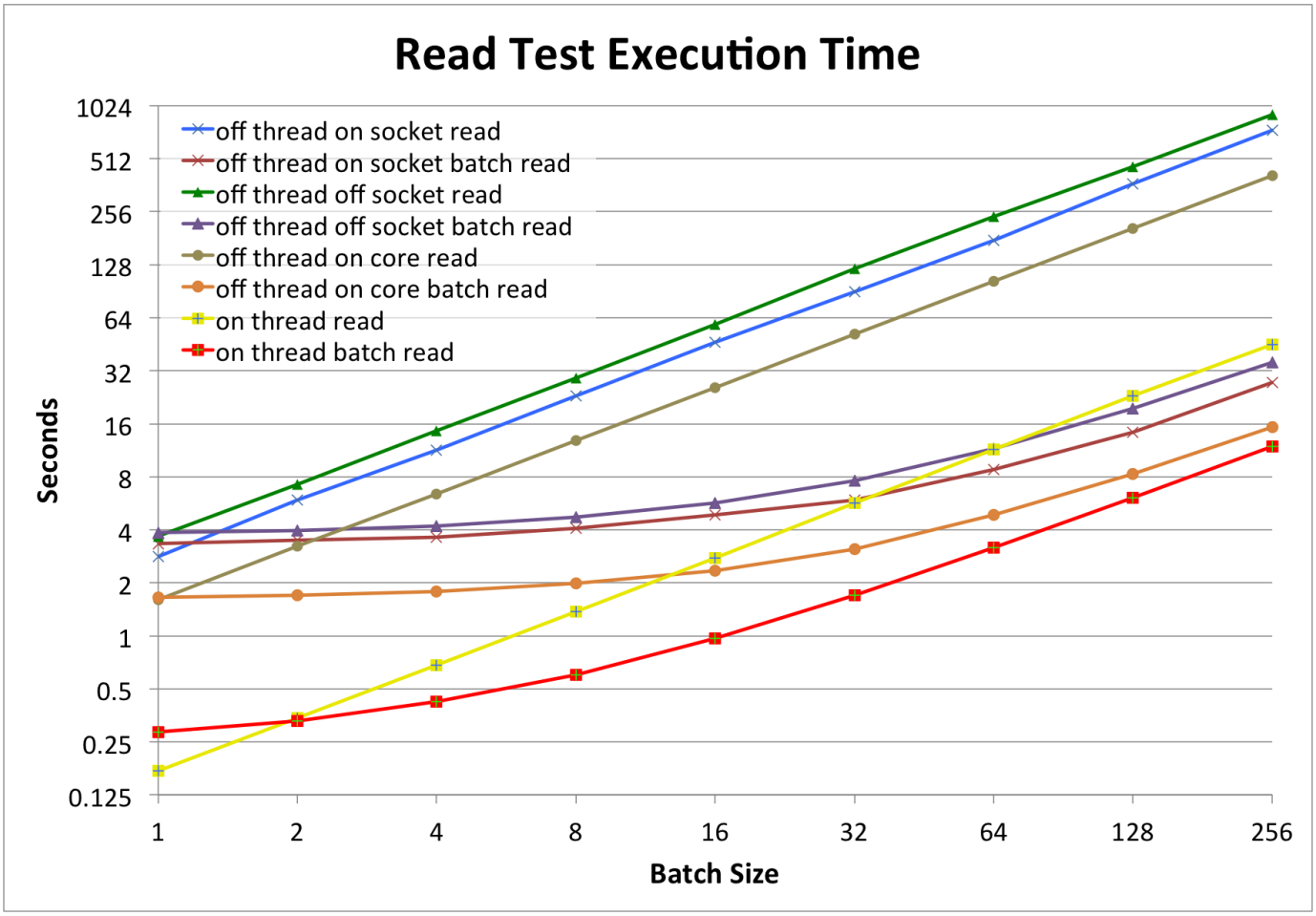


Fig. 5: Measurement of the execution time of one million batches comparing *pread* to the batch interface *ioctl* on a 2 socket 18 core Haswell node

Core and package C-state residency counters can also be accessed through functions in this part of the library.

Thermal– The Thermal interface can be used to access the per-core thermal sensors located on supported architectures. In addition, the TEMPERATURE_TARGET MSR can be viewed and modified. Possible applications of this MSR include thermal throttling to reduce cooling costs.

Integrated Memory Controller– The iMC on Intel processors contains many CSRs which act as performance counters for both the iMC and the DRAM [12][10]. Libmsr has simplified functions for calculating how much memory bandwidth is used. The memory bandwidth readings are obtained by using the iMC performance counters to count a certain event, and multiplying that by a fixed value. Thus, memory bandwidth is one of Intel's "derived events", not a direct register value [9]. Future work will include adding user-friendly functions for the most commonly used events and derived events.

V. Related Work

We hope that Libmsr, MSR-SAFE, and CSR-SAFE will catalyze HPC power and performance research. Our utilities will enable researchers to concentrate on higher level developments on more machines, instead of spending time learning architectural details and waiting for elevated privileges. Setting a one time power cap with RAPL is not the best solution for power-aware supercomputing. To meet the Department of Energy's requirement of an exascale computer under 20MW, a combination of over provisioning and dynamic power management is likely a necessity [2] [6] [5] [8]. A significant portion of modern power-aware research is focusing on dynamic power management. Davis et al. developed a software based utility named Star-Cap to dynamically manage the power of a cluster [3]. While this utility maintains a low overhead by using operating system performance counters, its response time may be as high as 20 seconds. This is unacceptable for HPC environments, where power spikes could damage the cluster or the power supply [5]. Star-Cap also requires power meters attached to the nodes, which we do not consider scalable. Hackenberg et al. used experiments to find the accuracy of the Fully Integrated Voltage Regulators (FIVRs) on Intel Haswell processors and determined them to have sufficient accuracy. Therefore, we see no reason for future HPC clusters to spend money on power meters.

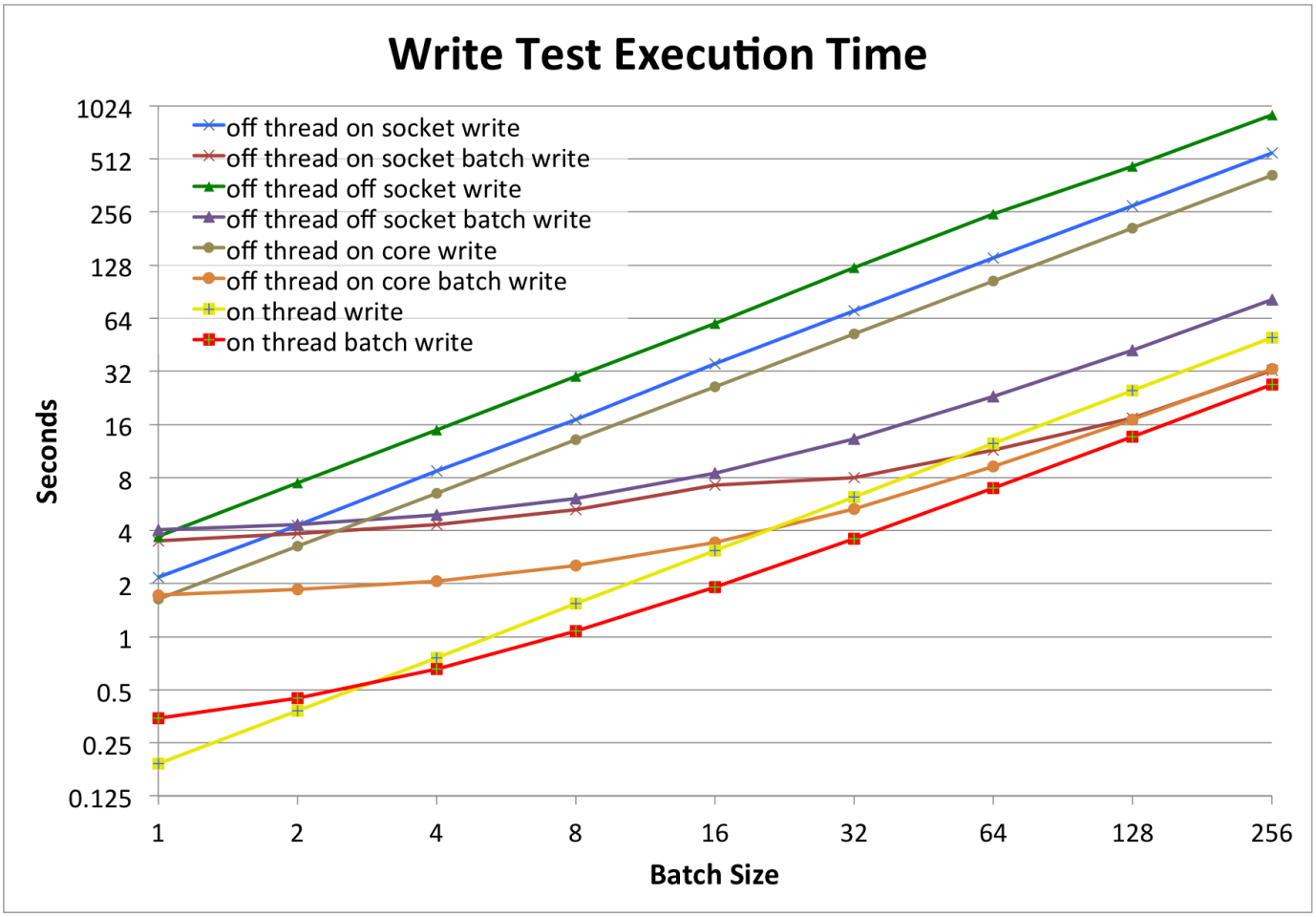


Fig. 6: Measurement of the execution time of one million batches comparing *pwrite* to the batch interface *ioctl* on a 2 socket 18 core Haswell node

Davis et al. mention that hardware counters have too much overhead to be viable for capping HPC power [3]. We believe that our recent optimizations in MSR-SAFE will now make hardware counters a more appealing candidate for scalable power management. Related research by Ellsworth et al. uses Libmsr to dynamically adjust power across an entire cluster while running various HPC jobs. Their experiments yielded a 14% increase in job throughput by redistributing waste power. Furthermore, overhead of the dynamic power management was negligible [5]. Fukazawa et al. used RAPL to place a power cap on the Magnetohydrodynamic (MHD) simulation code. They determined that the MHD code has certain sections where placing a processor power cap does not significantly effect performance [6]. Our demonstration of Libmsr's RAPL functionality on the NAS parallel benchmark MG confirms that a processor power cap does not significantly impact the performance of a memory bound application. We also showed that throttling memory power does have a significant effect on memory bound application performance as expected. In addition to power aware research, some recent studies also consider thermal aware computing [14]. The justification for this is that cooling costs are nearly half of the expenses for a data center. Zhao et al. developed a data center scheduler that was able to reduce cooling costs which utilized inlet temperature monitors [14]. The thermal MSRs interface in Libmsr may provide a less expensive, scalable alternative for thermal aware scheduling. Combs et al. used power meters attached to nodes to find the power signatures of HPC workloads [2]. They were able to identify applications based on their power signature with 85% accuracy even on different platforms and configurations. The node level power meter used in their study had the granularity of 1Hz. The MSR which contains energy readings of the processor has a granularity of 1KHz. We believe that this increased granularity would help improve the accuracy power signatures. We intend to investigate this in the future. Inadomi et al. developed a power budgeting algorithm which takes individual processor power variations into account [8]. They found that their algorithm scheduled applications that had a 1.8X average speedup when compared to a power budgeting algorithm which was not aware of processor power variations. Bailey et al. found that an optimal power reallocation algorithm combined with carefully

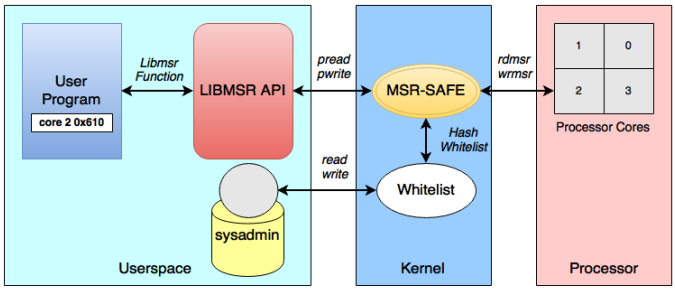


Fig. 8: User programs access MSRs through Libmsr which then uses the *pread* and *pwrite* calls in MSR-SAFE. MSR-SAFE ensures that the user's request has been whitelisted, ANDing off bits that aren't allowed or returning an error. If the user's request passes the whitelist check, then MSR-SAFE invokes a RDMSR or WRMSR instruction on the specified processor core and sends the data back to userspace.

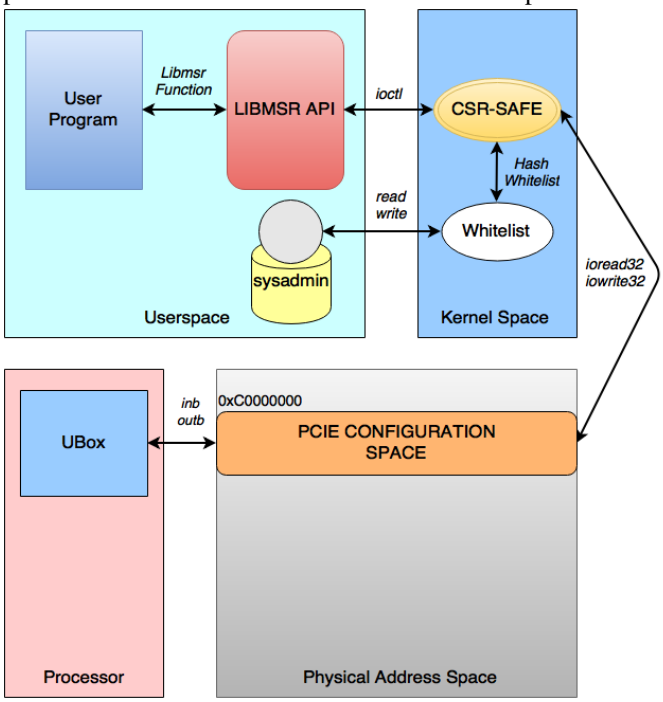


Fig. 9: Users access CSRs through Libmsr which uses the *ioctl* call in CSR-SAFE. CSR-SAFE checks the user's request against its own whitelist. CSR-SAFE has memory mapped the physical address space for the PCIe configuration space of whitelisted devices. Memory requests to these locations are routed to the UBox of the processor.

tuned OpenMP and MPI code can improve HPC algorithm performance by up to 41.1%.

VI. Conclusion and Future Work

Model specific registers and PCIe configuration space registers are useful for various HPC applications but are currently impractical with existing utilities. Our kernel modules MSR-SAFE and CSR-SAFE allow userspace access of CSRs and MSRs on Intel architectures. Our modules also maintain

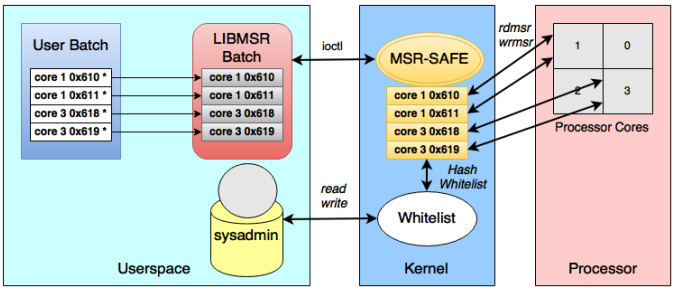


Fig. 10: An MSR batch requires that the user set up an array of references that is then directed to the actual data in Libmsr. This mechanism prevents needless data copying. After the array of operations reaches MSR-SAFE via ioctl, each operation is executed sequentially on the processor.

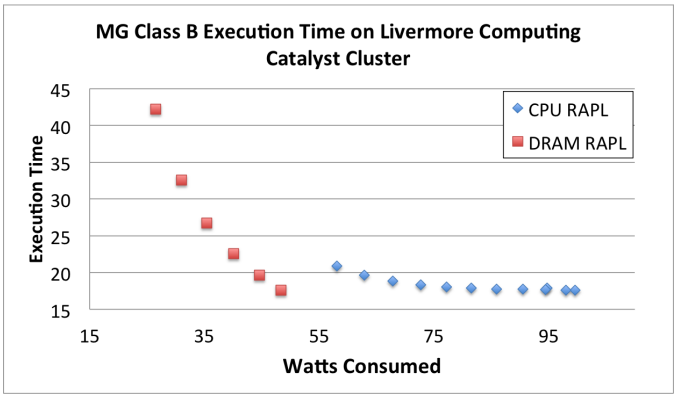


Fig. 11: Results from running the NAS Parallel Benchmark MG given various RAPL power limits in 5 watt increments. Since MG is memory bound, the execution speed does not vary greatly even with an extreme processor power cap. The execution speed does vary greatly when subjected to a memory power cap.

security thanks to the whitelist mechanism. Since the existing MSR kernel module has too much overhead to access large amounts of registers at high rates, we created an optimized way to access MSRs in batches with MSR-SAFE. The optimizations in the MSR-SAFE kernel module show a significant improvement over the existing MSR module, with speedups as high as 26X. Our Libmsr API solves usability issues of MSRs by providing a simplified interface for common tasks such as power management and performance monitoring. We have demonstrated a use case of Libmsr and MSR-SAFE by performing an experiment which shows that capping processor power can save a considerable amount of energy without significantly reducing performance for memory bound applications. Our PCIe configuration space module is the first of its kind to allow reading and writing all PCIe devices' CSRs in userspace. In the future we plan to add more CSR support to Libmsr, but this is a large undertaking because there are far more CSRs than MSRs and they change significantly between architectures. Although not discussed in detail in this

paper, Libmsr, MSR-SAFE, and CSR-SAFE have applications besides power-aware computing. Our utilities can also be used for performance monitoring, thermal monitoring and limiting, and control of various hardware settings and features. Future work includes using RAPL to evaluate an application power signature which can later aid in allocating power accordingly. Davis et al. believed that accessing hardware energy counters for this purpose was “more intrusive" than software counters [3], but we hope that our recent optimizations make the hardware energy counters a better solution. CSR accesses reach a bandwidth limit much faster than MSR accesses. We observed that our CSR operations which use IO remapping are faster than using the configuration read and write functions in the linux pci_dev struct. Finding the reason for this is something we hope to discover in subsequent experiments. Because processor manufacturers constantly add more registers and change existing ones, keeping up with these changes is also a priority. Finally, we hope to expand Libmsr, MSR-SAFE, and CSR-SAFE to support hardware from other vendors, including IBM, AMD, and Nvidia.

VII. Acknowledgements

We want to thank Dr. Barry Rountree for encouraging this research and providing suggestions for improvement. This material is based upon work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-682249).

References

[1] Peter E Bailey, Aniruddha Marathe, David K Lowenthal, Barry Rountree, and Martin Schulz. Finding the limits of power-constrained application performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 79. ACM, 2015.

[2] Jacob Combs, Jolie Nazor, Rachelle Thysell, Fabian Santiago, Matthew Hardwick, Lowell Olson, Suzanne Rivoire, Chung-Hsing Hsu, and Stephen W Poole. Power signatures of high-performance computing workloads. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 70–78. IEEE Press, 2014.

[3] John D Davis, Suzanne Rivoire, and Moises Goldszmidt. Star-cap: Cluster power management using software-only models. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 114–120. IEEE, 2014.

[4] NASA Advanced Supercomputing Division. NAS Parallel Benchmark Suite Version 3.3.1, 2015. https://www.nas.nasa.gov/publications/npb.html.

[5] Daniel A Ellsworth, Allen D Malony, Barry Rountree, and Martin Schulz. Dynamic power sharing for higher job throughput. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 80. ACM, 2015.

[6] Kyoko Fukazawa, Makoto Ueda, Masahiro Aoyagi, Tomonori Tsuhata, Kenta Yoshida, Aruta Uehara, Masakazu Kuze, Yuichi Inadomi, and Ken Inoue. Power consumption evaluation of an mhd simulation with cpu power capping. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 612–617. IEEE, 2014.

[7] Daniel Hackenberg, Robert Schone, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 896–904. IEEE, 2015.

[8] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 78. ACM, 2015.

[9] Intel. *Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual*, February 2014.

[10] Intel. *Intel Xeon Processor E5 v2 Product Family Datasheel Volume Two: Registers*, March 2014.

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual*, April 2015.

[12] Intel. *Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families Datasheet Volume 2: Registers*, June 2015.

[13] PAPI. Performance Application Programming Interface Version 5.4.3, 2016. http://icl.cs.utk.edu/papi/.

[14] Xiaogang Zhao, Tao Peng, Xiao Qin, Qiping Hu, Ling Ding, and Zhijun Fang. Feedback control scheduling in energy-efficient and thermal-aware data centers. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, 46(1):48–60, 2016.